

Can Languages and Tools Produce Better Software?

- In this talk, better means more reliable
 - Less likely to fail



SPT Approach

- Redundancy is good
- Redundancy exposes inconsistency

SPT Approach

- Redundancy is good
- Redundancy exposes inconsistency
- Inconsistency points to errors

SPT Approach

- Redundancy is good
- Redundancy exposes inconsistency
- Inconsistency points to errors
- Compare what programmer should do against what code does

Lightweight Specifications

- Start with description of correct/incorrect behavior
 - Not total specification \Rightarrow not total correctness
- Programmers provide specs
- Tools reward additional effort
 - Mechanically and systematically find errors

Programmers Write Types

- Successful, wide accepted specifications
 - Programmers understand and write type declarations
 - Declarations aid comprehension
 - Type checking is fast, routine, and finds errors
- Serve several roles
 - Documentation of interface syntax
 - Basis for program abstractions
 - Static (compile-time) error detection

Extend Successful Approach

- Specify and check other program properties
 - Languages (programming and domain-specific) express properties
 - Tools check that code obeys rules
- Goal is partial correctness
 - Detect and report important classes of errors
 - No guarantee of program correctness
- Systematic
 - Sound, complete analysis can flag absence of errors

SPT Overview

■ Reliable interfaces

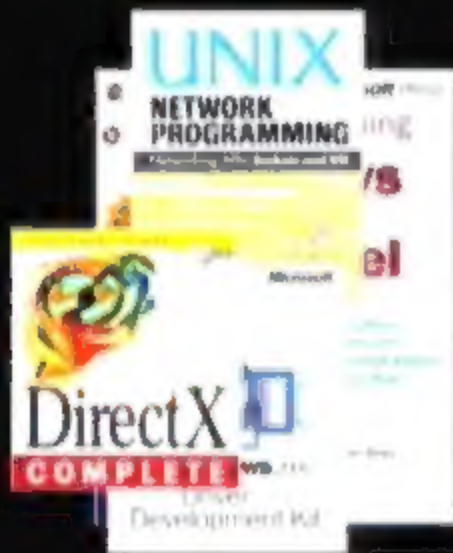
- Vault
- SLAM
- ESP

■ Concurrency and security

- Behave!
- Golf

Interfaces Have Usage Rules

- Specs in documentation
 - Incomplete, unenforced, wordy
 - Order of operations & data access
 - Resource management
- Disobeying rules causes bad behavior
 - System crash or deadlock
 - Unexpected exceptions
 - Failed runtime checks
 - Security flaws

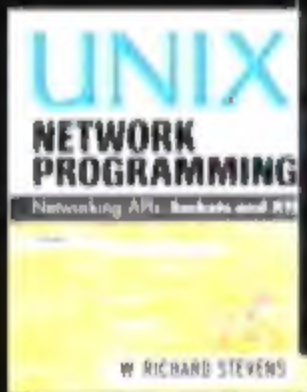


Example Usage Rule: Sockets

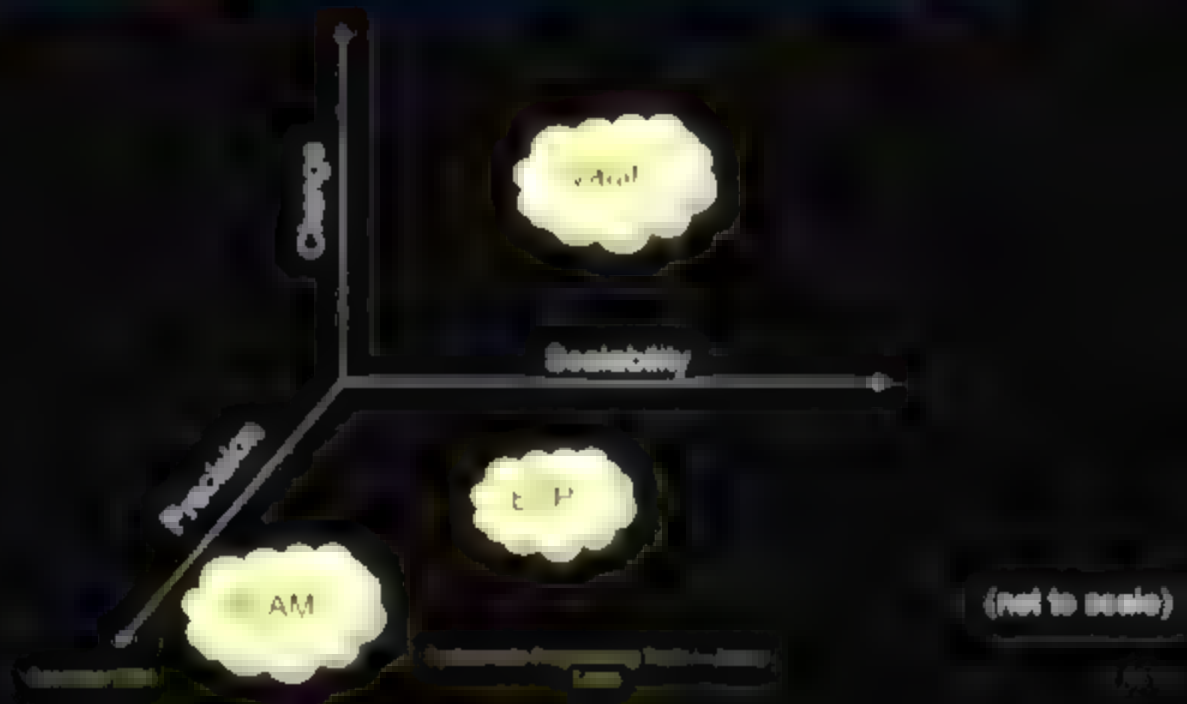
the "communication domain" in which communication is to take place; see `protocols(5)`.

Sockets of type `SOCK_STREAM` are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a `connect(2)` call. Once connected, data may be transferred using `read(2V)` and `write(2V)` calls or some variant of the `send(2)` and `recv(2)` calls. When a session has been completed a `close(2V)` may be performed. Out-of-band data may also be transmitted as described in `send(2)` and received as described in `recv(2)`.

The communications protocols used to implement a `SOCK_STREAM` insure that data is not lost or duplicated. If a piece of



SPT Specification & Checking



SPT Overview

■ Reliable interfaces

- Vault

- SLAM

- ESP

■ Security and concurrency

- Behavior

- Golf

Vault (Rob Delina, Manuel Fähndrich)



- Vault folds usage rules into programming language's type system
 - Interface author states rules in type signature
 - Compiler rejects client code that violates rule (type error)
 - Every violation guaranteed to be found (soundness)



Why a Type-based Approach?

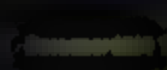
- Types are widely accepted form of specification
- Specification and code are kept in sync
- Developers find and fix bugs before execution
- Programmers understand violations and how to fix them
- Efficient, scalable analysis doesn't slow development



Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int) { S @ named → listening }
```



Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int) { S @ named → listening }
```




Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int) { S @ named → listening }
```

- Naming: **tracked types** give compile-time names (keys) to objects



Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int) { S @ named -> listening }
```

- Naming: **tracked types** give compile-time names (keys) to objects
- Object state: **key** can have symbolic state to model state of object



Tracked Types and Keys

- Change specs combine pre/post condition for functions



Tracked Types and Keys

- Change specs combine pre/post condition for functions

- Add key:

- `tracked(S) sock socket (...) [new S @ raw]`

- Consume key:

- `void close (tracked(S) sock) [-S]`

- Change key state:

- `void bind (tracked(S) sock) [S @ raw → named]`



Example: Sockets Interface

`Cracked() sock sockInit(hostname, style, port)`

`[new S & read]`

`void`

`bind(Cracked() sock, sockAddr)`

`[S & read -> connect]`

`void`

`listen(Cracked() sock, port)`

`[S & connect -> Networking]`

`Cracked() sock accept(Cracked() sock)`

`[S & Networking,
new S & ready]`

`void`

`connect(Cracked() sock, byte[]) [S & ready]`

`void`

`close(Cracked() sock)`

`[~S]`



Example: Sockets Interface

trackd() sock socket(domain, type, int)

[new S & read]

void

bind(trackd()) sock, struct

[S & read -> named]

void

listen(trackd()) sock, int

[S & named -> Netmng]

trackd() sock **accept(trackd())** sock

[S & Netmng,

new S & ready]

void

read(trackd()) sock, byte[] [S & ready]

void

write(trackd()) sock

[-af]



Statically Enforcing Usage Protocols

- At every program point, Vault compiler computes key set
 - Functions and built-in operations (new/free) change key set
 - Key must be consumed, not "leaked"
- On each path in function's body, check:
 - Pre-condition transformed into post-condition
 - All proof obligations satisfied
 - Pre-conditions of other function calls
 - Primitive operations (memory access, free)
- Avoid exponential blow-up
 - Require uniform predicate at join points



Checking Socket Client Code

```
void summarize(socket_t addr) {  
    created(&) sock s = socket("tcp", "server", 0);  
    bind(s, addr);  
    listen(s, 0);  
    while (1) {  
        created(&) sock c = accept(s);  
        receive(s, buffer);  
        close(c);  
    }  
    close(s);  
}
```



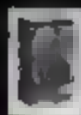

Checking Socket Client Code

```
void summarize(sockaddr addr) {  
    tracked() sock s = socket("tcp", "stream,0");  
    bind(s, addr);  
    listen(s, 0);  
    while (1) {  
        tracked() sock c = accept(s);  
        receive(c, buffer);  
        close(c);  
    }  
    close(s);  
}
```



Checking Socket Client Code

```
void summarize(socketaddr addr) {  
    tracked(&) sock s = socket("tcp", "stream,0);  
    bind(s, addr);  
    listen(s, 0);  
    while (intention()) {  
        tracked(&) sock c = accept(s);  
        receive(s, buffer);  
        close(c);  
    }  
    close(s);  
}
```



Checking (Wrong) Socket Client Code

```
void connect_to(sockaddr addr) {  
    checked() sock s = socket("tcp", "stream", 0);  
    bind(s, addr);  
    listen(s, 0);  
    while (1) {  
        checked() sock c = accept(s);  
        receive(c, buffer);  
        close(c);  
    }  
    close(s);  
}
```



Checking (Wrong) Socket Client Code

```
void summarize(socket_t addr) {  
    struct sockaddr_in s;  
    bind(s, addr);  
    listen(s, 0);  
    while (1) {  
        struct sockaddr_in c;  
        receive(c, buffer);  
        close(c);  
    }  
    close(s);  
}
```



Checking (Wrong) Socket Client Code

```

void communicate(socketaddr addr) {
    struct sockaddr_in sock;
    bind(s, addr);
    listen(s, 0);
    while (1) {
        struct sockaddr_in sock;
        receive(s, buffer);
        close(s);
    }
    close(s);
}

{ }
{ struct sockaddr_in }
{ listening }
{ listening }
{ listening, 3ready }
{ listening, 3ready }
{ 3ready }
{ }
{ }

```

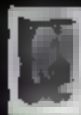
✗ No agreement
at join point



Checking (Wrong) Socket Client Code

```
void communicate(socketaddr addr) {  
    tracked() sock s = socket("tcp", "stream", 0);  
    bind(s, addr);  
    listen(s, 0);  
    while (listening()) {  
        tracked() sock c = accept(s);  
        receive(c, buffer);  
        close(c);  
    }  
    close(s);  
}
```

Listing 4.1



Checking (Wrong) Socket Client Code

```
void summarize(socketaddr addr) {  
    struct sockaddr sock = socket('tcp', SOCK_STREAM, 0);  
    bind(sock, addr);  
    listen(sock, 5);  
    while (1) {  
        struct sockaddr sock2 = accept(sock);  
        receive(sock2, buffer);  
        close(sock2);  
    }  
    close(sock);  
}
```

✗ **Uninitialized variable**



Vault Research

- Few new language features express and check many usage rules
 - Still imperative programming style
 - Handle "real" applications: device drivers, DirectX
 - Check "real" rules that developers commonly violate
- On-going research
 - Usage rules in C#
 - Safe, explicit memory management in CLR

SLAM

(Tom Ball, Srinivas Rajaman)

■ Input

- C source code, "as is"
- API rules in SLIC language

■ Automatically create abstraction of C program

- Abstract model = Boolean program

■ Systematic exploration of model's state space

- Does feasible path through program lead to error state in SLIC?

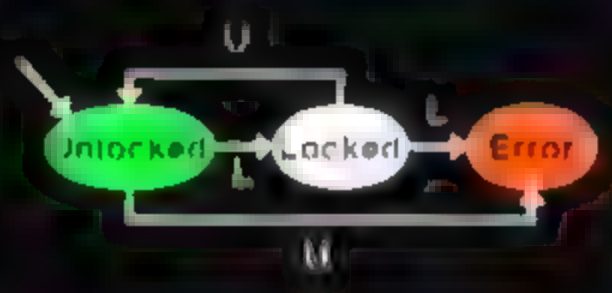
■ Demand-driven refinement of model

- Exclude infeasible paths



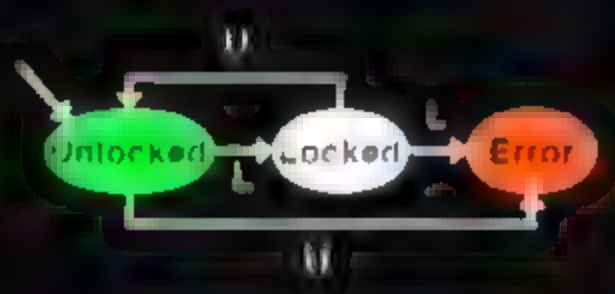
API Rule for Locking

State Machine



API Rule for Locking

State Machine



SUC

```
state {
    int locked = 0;
```

```
lock.call {
    if (locked == 1) abort;
    else locked = 1;
```

```
unlock.call {
    if (locked == 0) abort;
    else locked = 0;
```




State-based Search Fails

```
do {  
    //get the write lock  
    Lock.lock();  
    ReleaseAndSpinLock(&driver->writeLock);  
    if (packets[0] != packets[1])  
        request = driver->writeLockId;  
    if (request < request->start) {  
        driver->writeLockId = request->start;  
        Unlock.lock();  
        ReleaseAndSpinLock(&driver->writeLock);  
        ...  
        packets++;  
    } while (packets != packets+1);  
    Unlock.lock();  
    ReleaseAndSpinLock(&driver->writeLock);  
}
```

State-based Search Fails

```

//get the write lock
lock.call();
if(!request.isSpunLock(&devInt->writeLock)) {
    if(packetOld == packet) {
        request = devInt->writeLock.request();
    }
    if (request == request->parent) {
        devInt->writeLock.isSpunLock = request->parent;
        lock.call();
        if(!request.isSpunLock(&devInt->writeLock)) {
            ...
        }
    }
    while (!packet == packetOld) {
        lock.call();
        if(!request.isSpunLock(&devInt->writeLock)) {

```


State-based Search Fails



```
do {
    //get the write lock
    Lock.acquire(lock);
    Release(lock);

    nPacketsOld = nPackets;
    request = devInt->GetWriteHead();

    if (request < request->total) {
        devInt->WriteHeadWrite = request->Start;
        Lock.acquire(lock);
        Release(lock);
        ...
        nPackets++;
    } while (nPackets != nPacketsOld);
    Unlock.acquire(lock);
    Release(lock);
}
```

State-based Search Fails

```

do {
    //get the write lock
    Lock.acquire();
    ReleaseWriteLock(ddevRef->getWriteLock());

    nPacketsOld = nPackets;
    request = devRef->getTailPacket();

    if (request < request->getLen()) {
        devRef->getTailPacket() = request->getLen();
        Lock.acquire();
        ReleaseWriteLock(ddevRef->getWriteLock());
        ...
        nPackets++;
    } while (nPackets != nPacketsOld);
    Unlock.acquire();
    ReleaseWriteLock(ddevRef->getWriteLock());
}

```

State-based Search Fails

```
do {
    //get the write lock
    Lock.mall();
    HoldSemaphoreLock(&drvWrite-SemaphoreLock);

    nPacketsOld = nPackets;
    request = drvWrite->WriteReqHandle();

    if (request <= request->Priority) {
        drvWrite->WriteReqHandle = request->Priority;
        Unlock.mall();
        HoldSemaphoreLock(&drvWrite-SemaphoreLock);
        ...
        nPackets++;
    }
} while (nPackets != nPacketsOld);
Unlock.mall();
HoldSemaphoreLock(&drvWrite-SemaphoreLock);
```

State-based Search Fails

```
do {
    //get the write lock
    Lock.call();
    Mutex.unlock(&devLib->writeLock);

    nPacketsOld = nPackets;
    request = devLib->Write(&nPackets);

    if (request < request->status) {
        devLib->Write(&nPackets) = request->status;
        Unlock.call();
        Mutex.unlock(&devLib->writeLock);
        ...
        nPackets++;
    } while (nPackets != nPacketsOld);
    Unlock.call();
    Mutex.unlock(&devLib->writeLock);
}
```

State-based Search Fails

```
do {
    //get the write lock
    Lock.mall();
    ReleaseSpinLock(&devInt->writeLock);

    nPacketsOld = nPackets;
    request = devInt->WriteReqStHead();

    if (request && request->status) {
        devInt->WriteReqStHead = request->Next;
        Unlock.mall();
        ReleaseSpinLock(&devInt->writeLock);
        goto nPackets++;
    }
} while (nPackets != nPacketsOld);
Unlock.mall();
ReleaseSpinLock(&devInt->writeLock);
```

State-based Search Fails



```
do {
    //get the write lock
    Lock.mll();
    HdrQueue.enqueueLock(&devHdr->writeLock);

    nPacketsOld = nPackets;
    request = devHdr->writeLockHdr;

    if (request && request->status) {
        devHdr->writeLockHdr = request->next;
        Lock.mll();
        HdrQueue.enqueueLock(&devHdr->writeLock);
        ...
        nPackets++;
    }
} while (nPackets != nPacketsOld);
Unlock.mll();
HdrQueue.enqueueLock(&devHdr->writeLock);
```

State-based Search Fails



```
//get the write lock:
Lock.lock();
//request to get lock (if driver is not holding lock)
request = driver->RequestLock();

nPacketsOld = nPackets;
request = driver->RequestLock();

if (request < request->status) {
    driver->RequestLock() = request->status;
    Unlock.lock();
    //Release lock (if driver is not holding lock)
    ...
    nPackets++;
}

while (nPackets != nPacketsOld) {
    Unlock.lock();
    //Release lock (if driver is not holding lock)
    ...
}
```

State-based Search Fails



```

do {
    //get the write lock
    Lock.mall();
    Mutex.acquireLock(&devInt->writeLock);

    if (state == 0) { // nPacket == 0
        request = devInt->writeLockInfo;

        if (request < request->start) {
            devInt->writeLockInfo = request->start;
            Lock.mall();
            Mutex.acquireLock(&devInt->writeLock);
            ...
            nPackets++;
        }
    } while (nPackets != nPacketsOld);
    Unlock.mall();
    Mutex.releaseLock(&devInt->writeLock);
}

```


Boolean Program; Abstraction 1

```

do
  //get the write lock
  lock.acquire()
  if (*) then
    unlock.acquire()
  if (*) then
  else
  fi
fi
while (*)
  lock.release()

```



Boolean Program; Abstraction 1

```
do
  //get the write lock
  lock.call();
  if (*) then
    unlock.call();
  if (*) then
  else
  fi
fi
while (*);
lock.call();
```

Boolean Program; Abstraction 1



```

do {
  //get the write lock
  lock.call();
  if (*) then
    unlock.call();
  if (*) then
  else
    fl
  fl
  while (*);
  lock.unlock.call();
}
  
```

Boolean Program: Abstraction 1



```
do {
    //get the write lock
    lock.call();

    if (!zoom)
        unlock.call();

    if (!zoom)
        else
        {
            //
        }

    while (!);
    //lock.call();
}
```

Boolean Program; Refine Abstraction

- Examine infinite path
- Introduce boolean variable b to represent:
 $\#PacketsOld == \#Packets$

```
do
  // get the write lock
  lock.call();

  #packetsOld = #packets;

  if (x) then
    unlock.call();

    if (y) then
      ...
    fi
    #packets++;
  fi

  while (callPackets != #packetsOld)
    unlock.call();
```

Boolean Program; Refine Abstraction

- Eliminate infeasible path
- Introduce boolean variable b to represent:
 $nPacketsOld == nPackets$

```

do
  //get the write lock
  lock.call()

  nPacketsOld = nPackets;
  if (0) then
    unlock.call()
  else if (0) then
  else
    fi
    nPackets++;
  fi
while (nPackets != nPacketsOld)
unlock.call()

```

Boolean Program; Refine Abstraction

- Examine infeasible path
- Introduce boolean variable b to represent:
 $\#PacketsOld == \#Packets$

```
do
  //get the write lock
  lock.acquire();

  b = true;
  if (!b) then
    lock.release();

  if (!b) then
  else
  fi;

  b := b ? false : true;
fi;
```

```
fi;
while (!b)
  lock.acquire();
fi;
```

Boolean Program; Abstraction 2

```

do
  //get the write lock
  Lock.call(lb)
  b := true;
  if (*) then
    Unlock.call(lb)
  if (*) then
  else
    fi
    b := b ? true; (*)
  fi
while ( lb > 0 )
Unlock.call(lb)

```


Boolean Program; Abstraction 2



```
//get the write lock
lock.acquire(1);

do {
    //do something
} while (true);

//release the lock
lock.release();
```

Checking API Rules with SLAM



SLAM Summary

- Automatic analysis (no code annotation)
 - SLIC specification of API rules
- Report errors with source-level paths
- Very precise analysis
 - Minimize false positives
- Current research
 - Applying to device drivers
 - Tool scalability

The Static Driver Verifier

Thomas R. H.
Sriram K. Rajaman

Software Productivity Tools

Programmer Productivity
Research Center

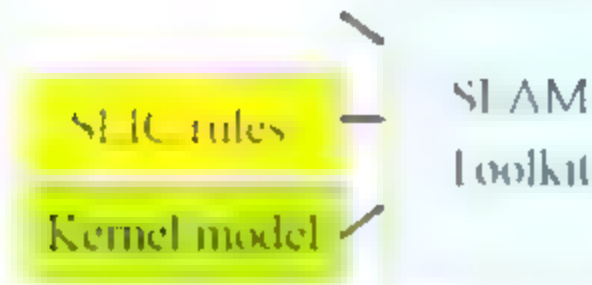
The Static Driver Verifier

Thomas Ball
Sriram K. Rajamani

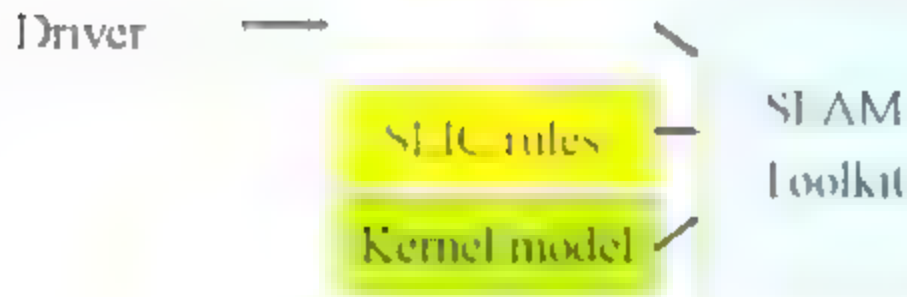
Software Productivity Tools

Programmer Productivity
Research Center

Static Driver Verifier



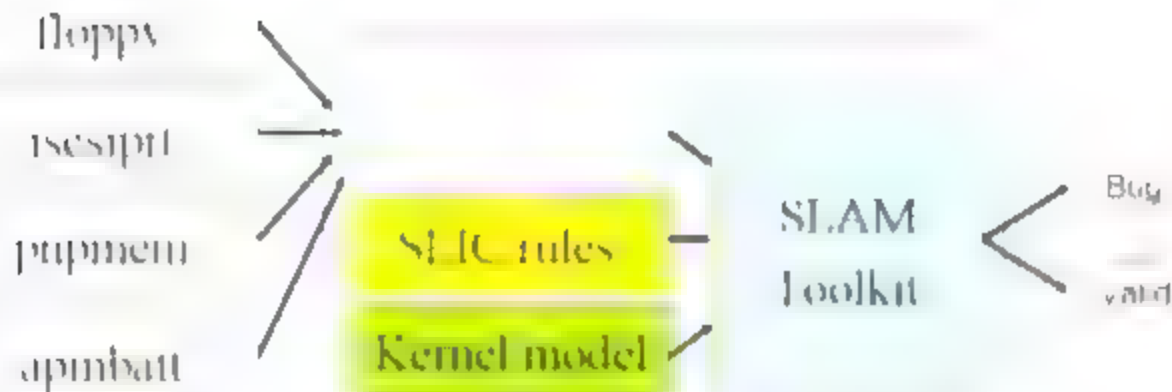
Static Driver Verifier



Static Driver Verifier

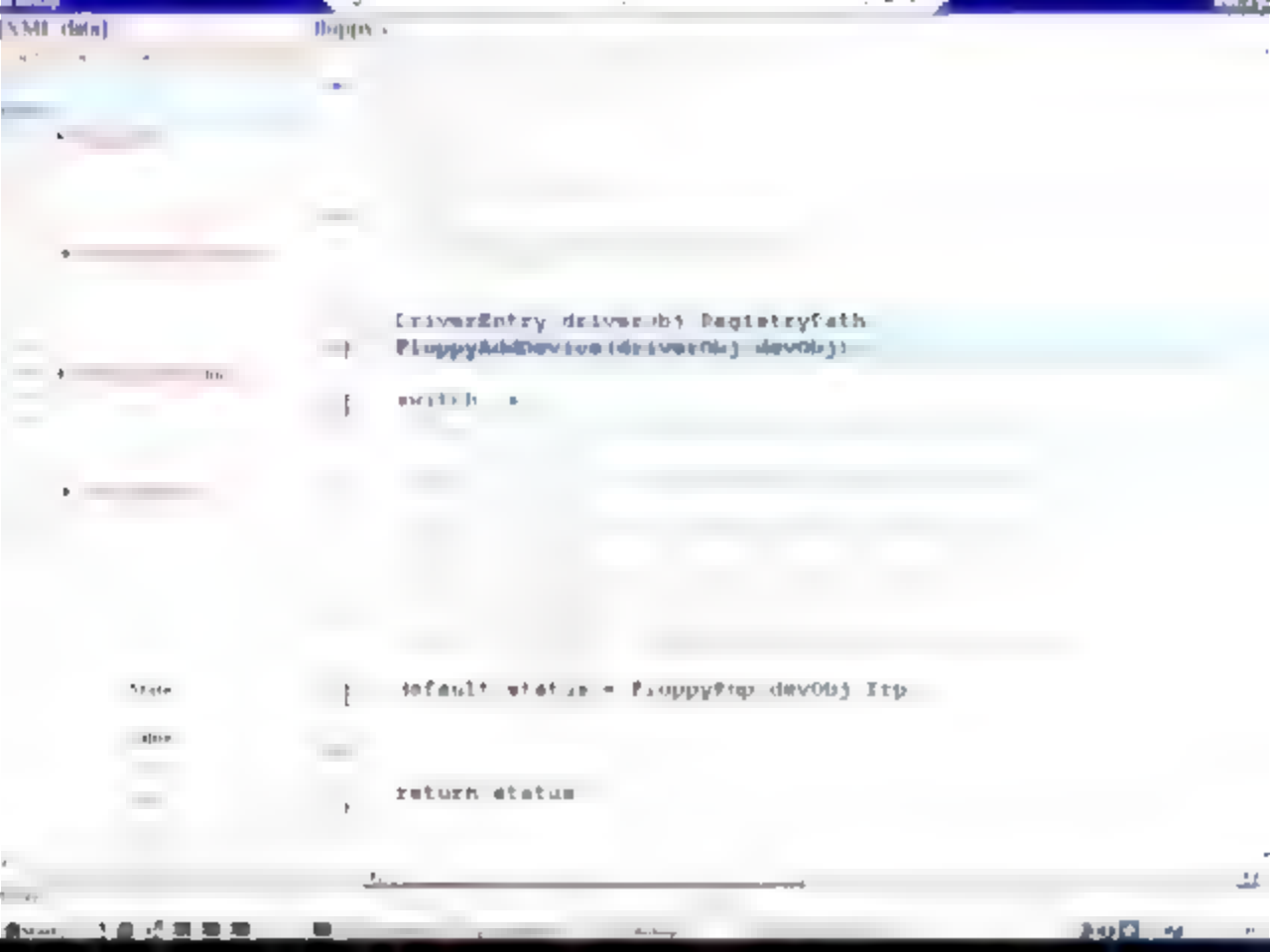


Static Driver Verifier (ii)







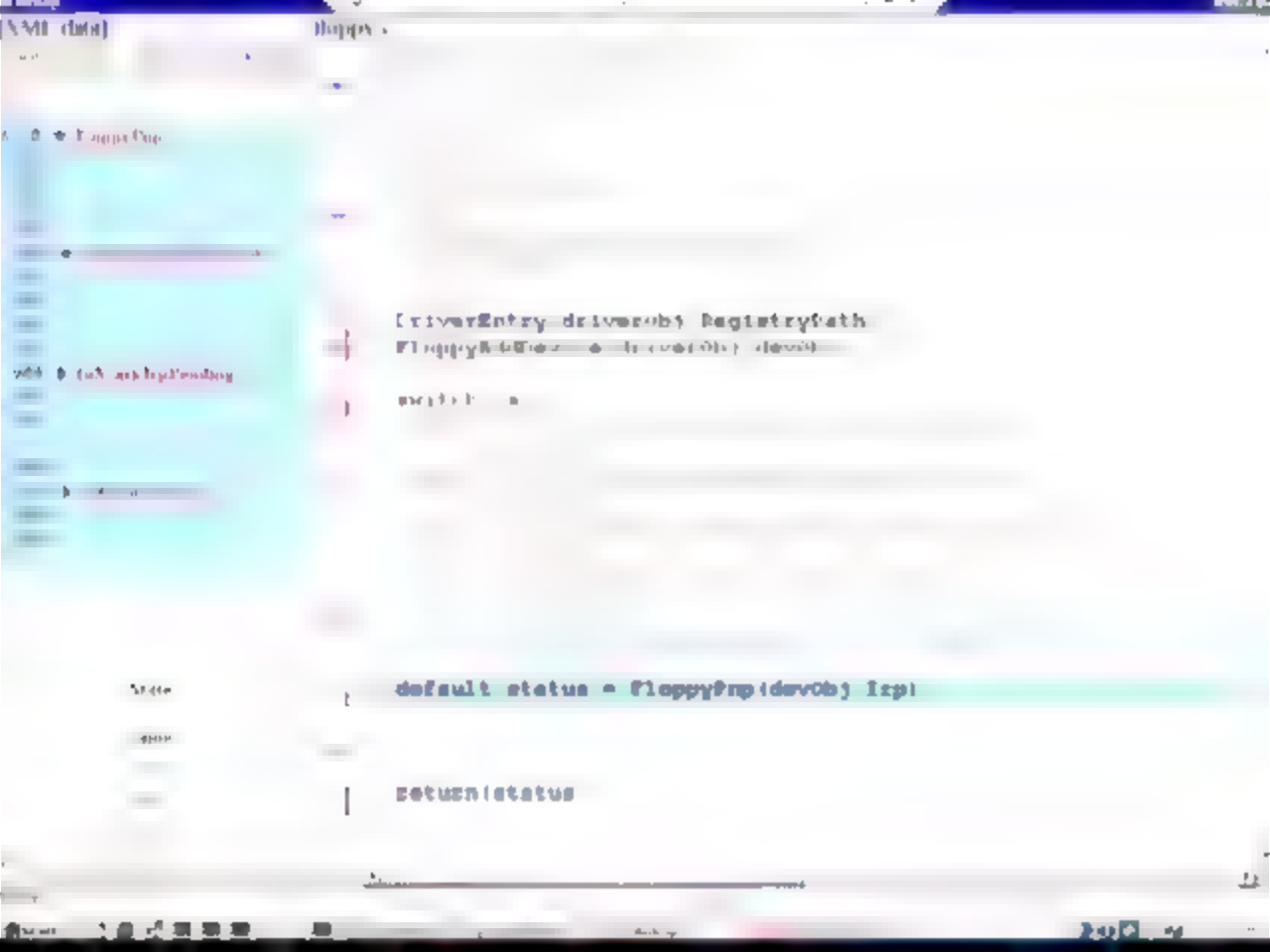


```
driverEntry (driver) RegistryPath  
PluggableDevice (driver) devObj)
```

```
{  
    switch (
```

```
default: status = PluggableDevice (devObj) Itp
```

```
return status
```



```

    def __init__(self, disketteExtension, HoldNewerDiskette):
        self.disketteExtension = disketteExtension
        self.HoldNewerDiskette = HoldNewerDiskette

```

```

    ntstatus = WinUsbIrpToThread(Irp, disketteExtension)

```

```

    if ntstatus == STATUS_PENDING:

```

```

        ResultForming, object = disketteExtension.FloppyThread

```

```

    if disketteExtension.FloppyThread != None:

```



```

    kReleaseofsetMutex &DisketteExtension ThreadReferenceMu

```

```

    ToMarkTpending(Tp)

```

```

    KlistarlockedinertTallList

```

```

    kReleaseofsetMutex

```

```

    return STATUS_PENDING

```

State

State

ExclusioefsetMutex &DisketteExtension ThreadReferenceMu

IoMethIrppending.Irp

KernelIoCompletionTailList

KeReleaseSemaphore

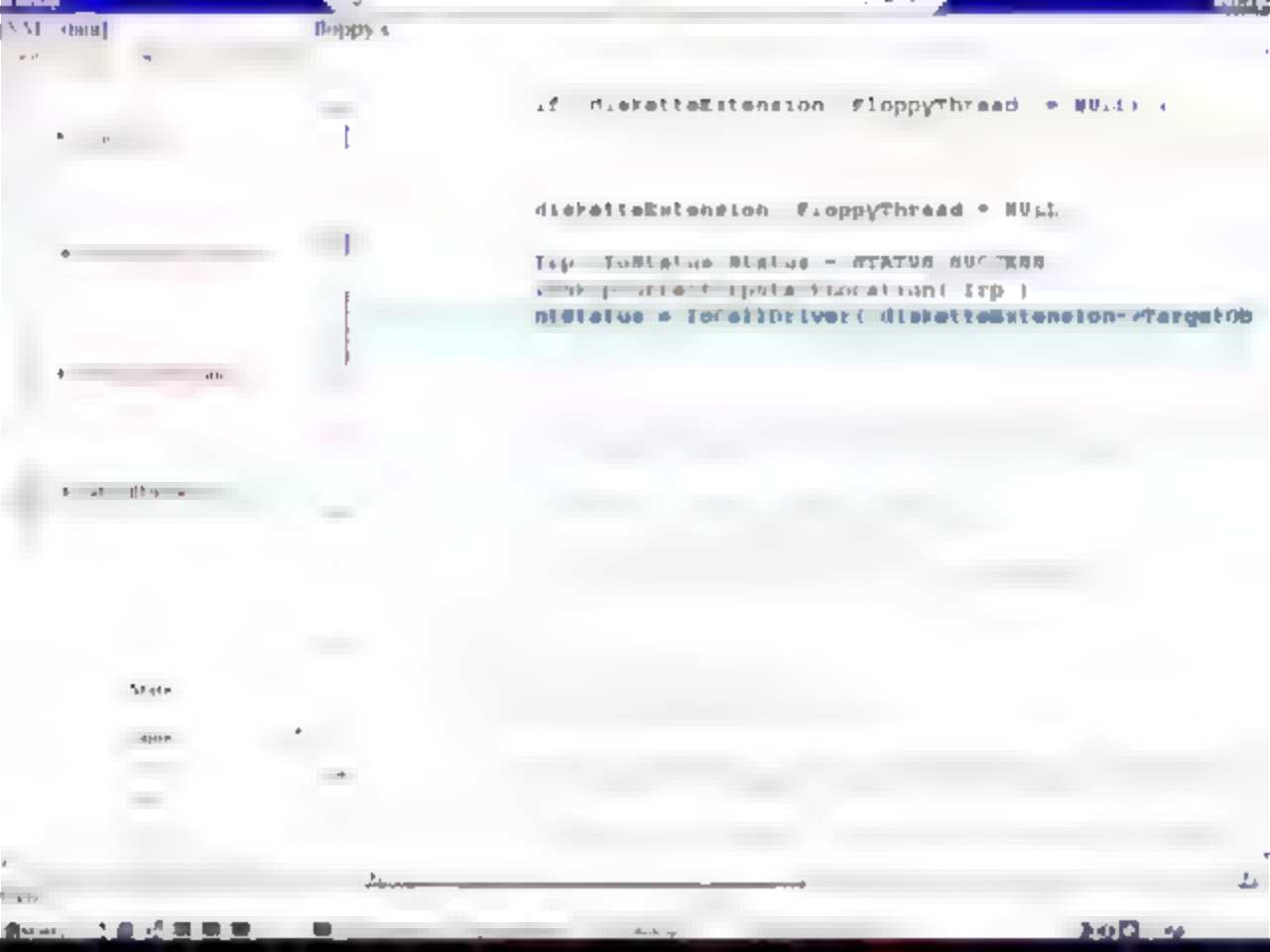
return STATUS_PENDING

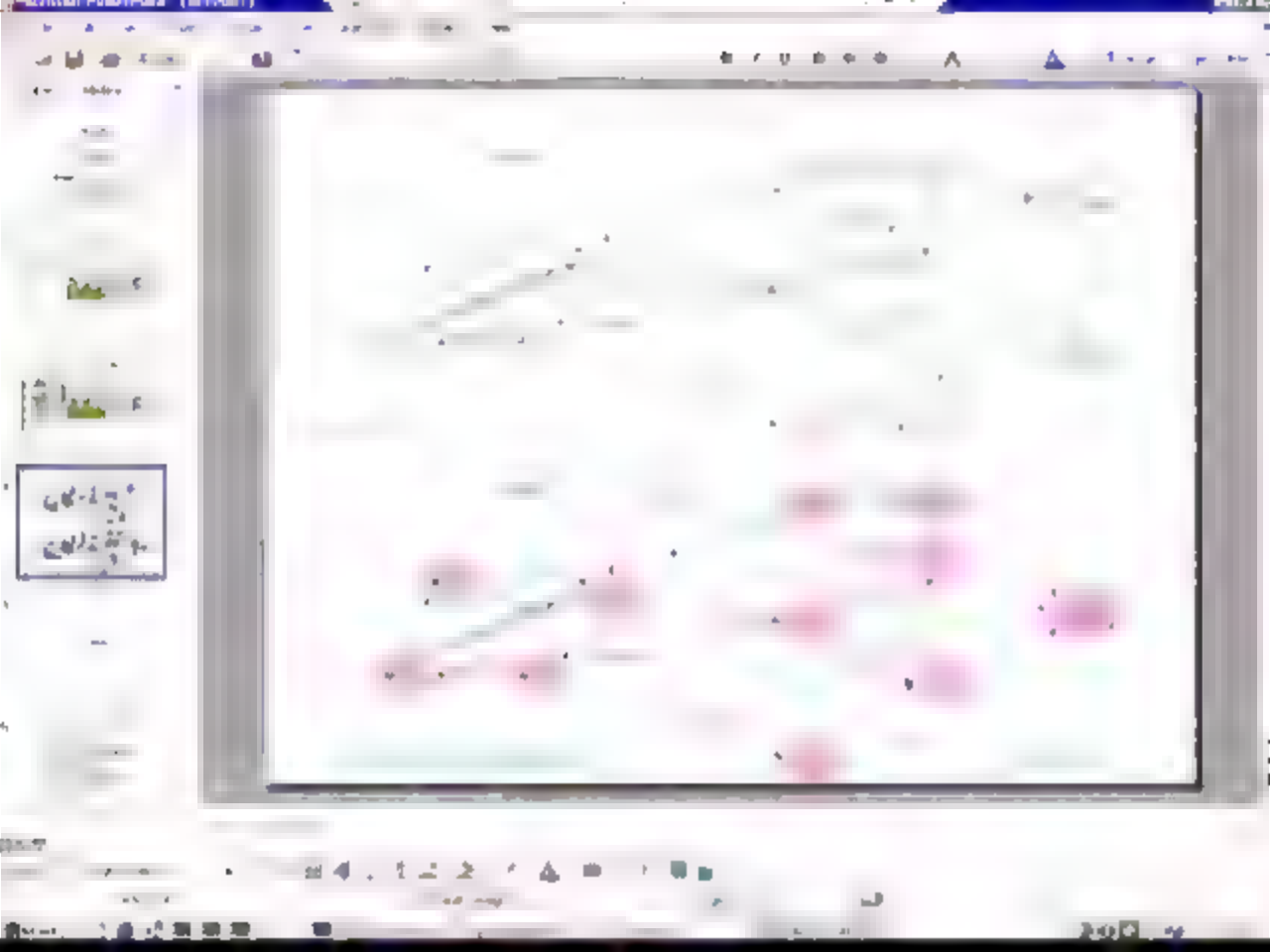
NTFS

NTFS

NTFS

NTFS





ESP (Manuvir Das)

- Error detection via scalable program analysis
- Similar to SLAM:
 - SLIC-like specification
 - Automatic analysis
 - Sound (finds all errors)
- Differences from SLAM:
 - Targeted at large (million+ LOC) C++ code bases
 - Only parameterized finite state protocols
 - Weaker (less precise) analysis

Parameterized Protocols

- FSM attached to data items
- Syntactic patterns trigger transitions



Get code pattern	Transition
<code>o->pin_spinlock.Get()</code>	<code>Lock(e)</code>
<code>o->pin_spinlock.Release()</code>	<code>Unlock(e)</code>
<code>return o</code>	<code>Ret</code>

ESP Insight

- Three distinct entities
 - Sequence of actions along control flow path
 - Data involved in actions
 - `e->Get(); p = e; p->Release();`
 - Data involved in path feasibility
 - `if (c) e->Get();` `if (c) e->Release();`
- Insight: use different static analyses to track each entity

Three Phases of Static Analysis

1. Global value flow analysis
 - Use GOLF to build call and value flow graph
2. Global protocol tracking
 - Use dataflow and value flow to track protocols associated with data items
3. Local feasibility analysis
 - Use local abstract simulation + summaries from Phase II to rule out infeasible paths



Example From SQL

```
void LockObj::Free() {  
    m_spinlock.Unlock();  
  
    if (used) {  
        wprintf(L"Freeing lock\n");  
        return;  
    }  
  
    m_spinlock.Release();  
}  
  
void wprintf(Lock *lock) {  
    lock->Print();  
}
```

Example From SQL

```
void LockObjent::Free() {
    m_spinlock.Get();

    if (used) {
        unlockobj(m_spinlock);
        return;
    }

    m_spinlock.Release();
}
```

```
void unlockobj(Lock *lock) { Phase 1
    lock->Unlock(); } // Release called on pointer target of first parameter
```

Phase 2 —
 —> summary of FGM behavior
 "part : Locking => lock, lock => Over"



Example From SQL

```
void LockObj::Free() {  
    m_spinlock.Lock();  
  
    if (used) {  
        unlockobj (&m_spinlock);  
        return;  
    }  
  
    m_spinlock.Release();  
}
```

Phase 1

- target of unlockobj is m_spinlock
- Lock and Release called on m_spinlock

```
void unlockobj(Lock *lock) {  
    lock->Unlock();  
}
```

Phase 1

- Release called on pointer target of first parameter

Phase 2

- Dataflow summary of FWH between
"var1 : Locked => lock, lock => Free"

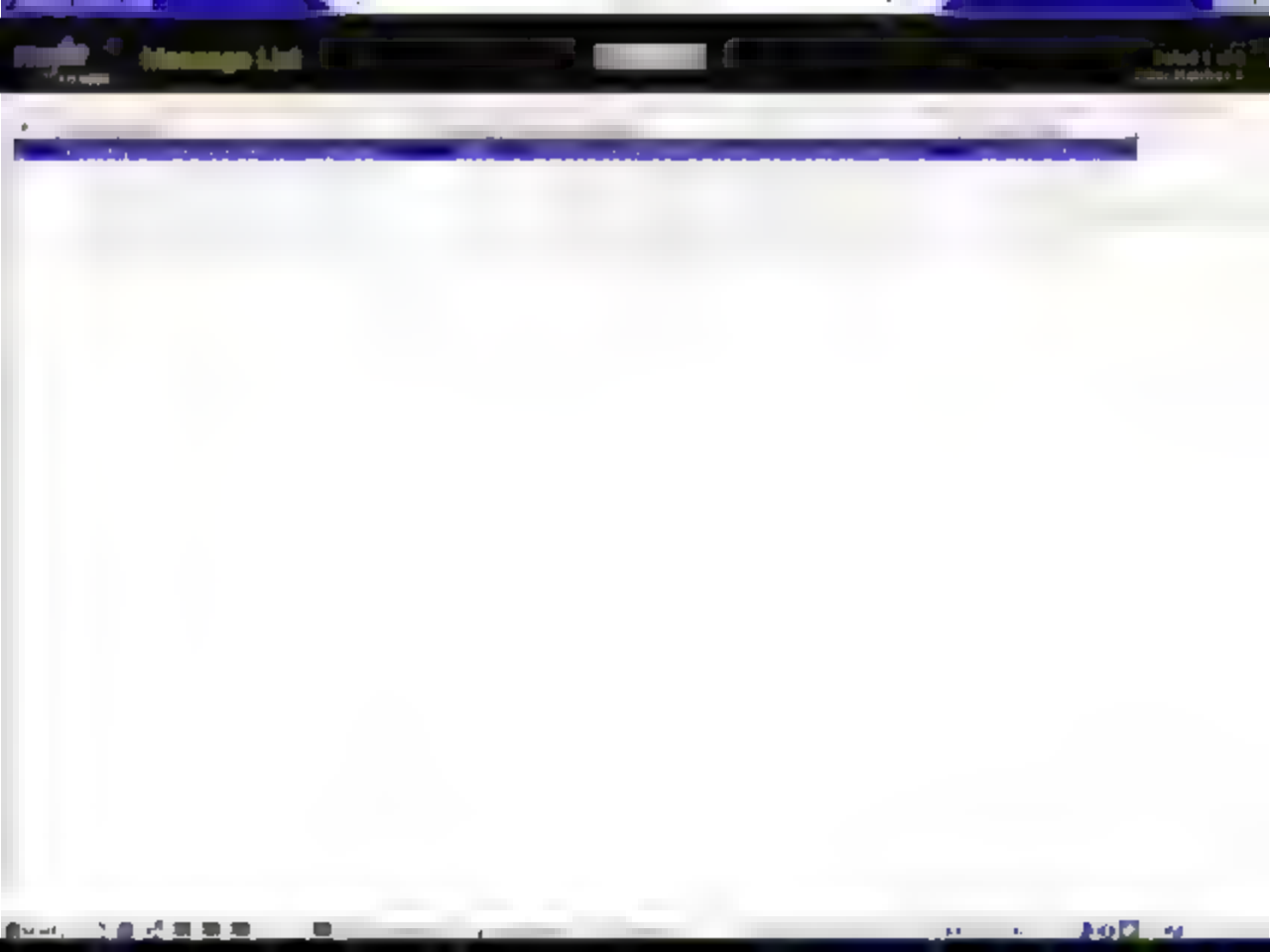
Unlocked (Summary Value) = Free

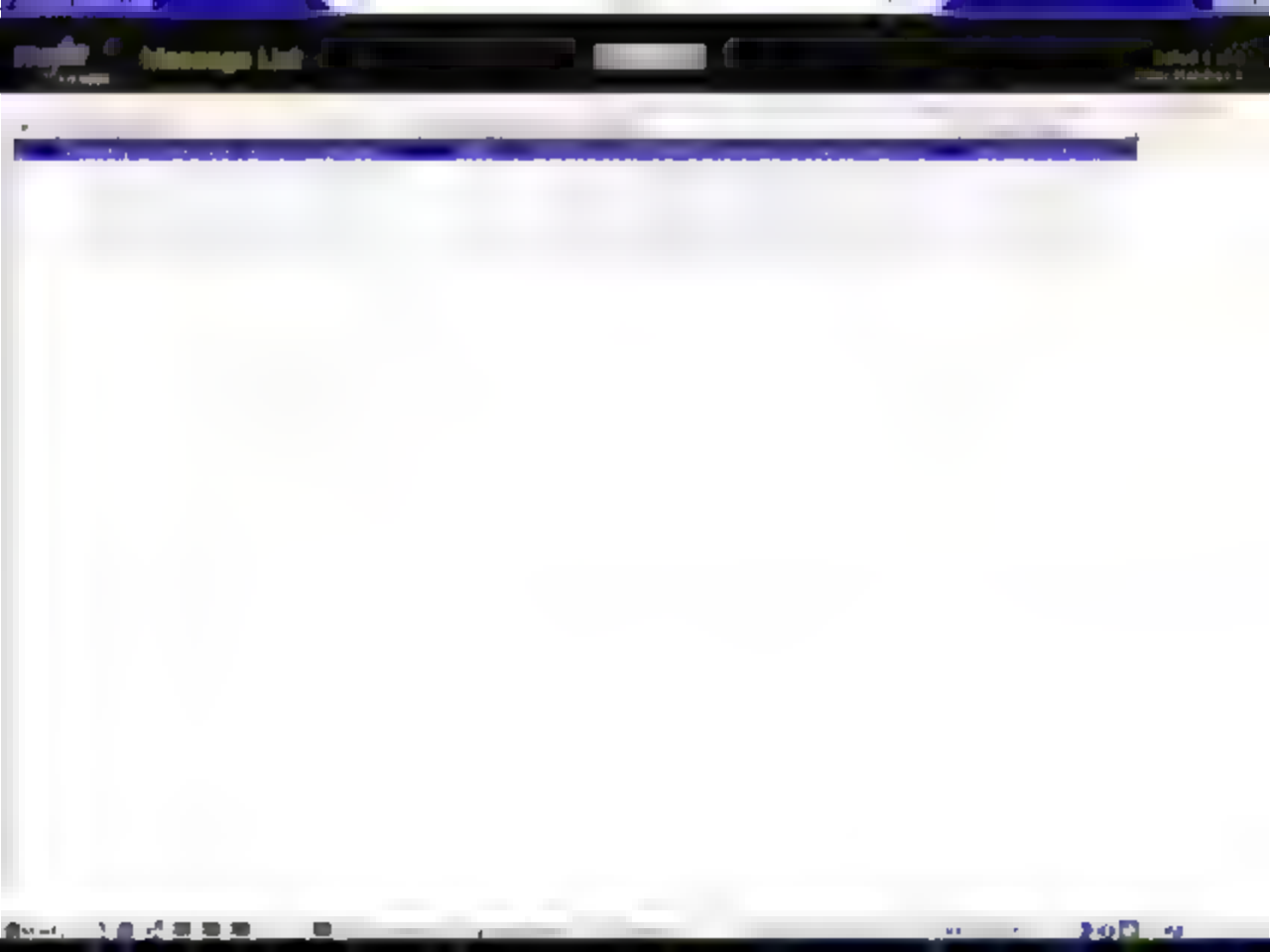
lock

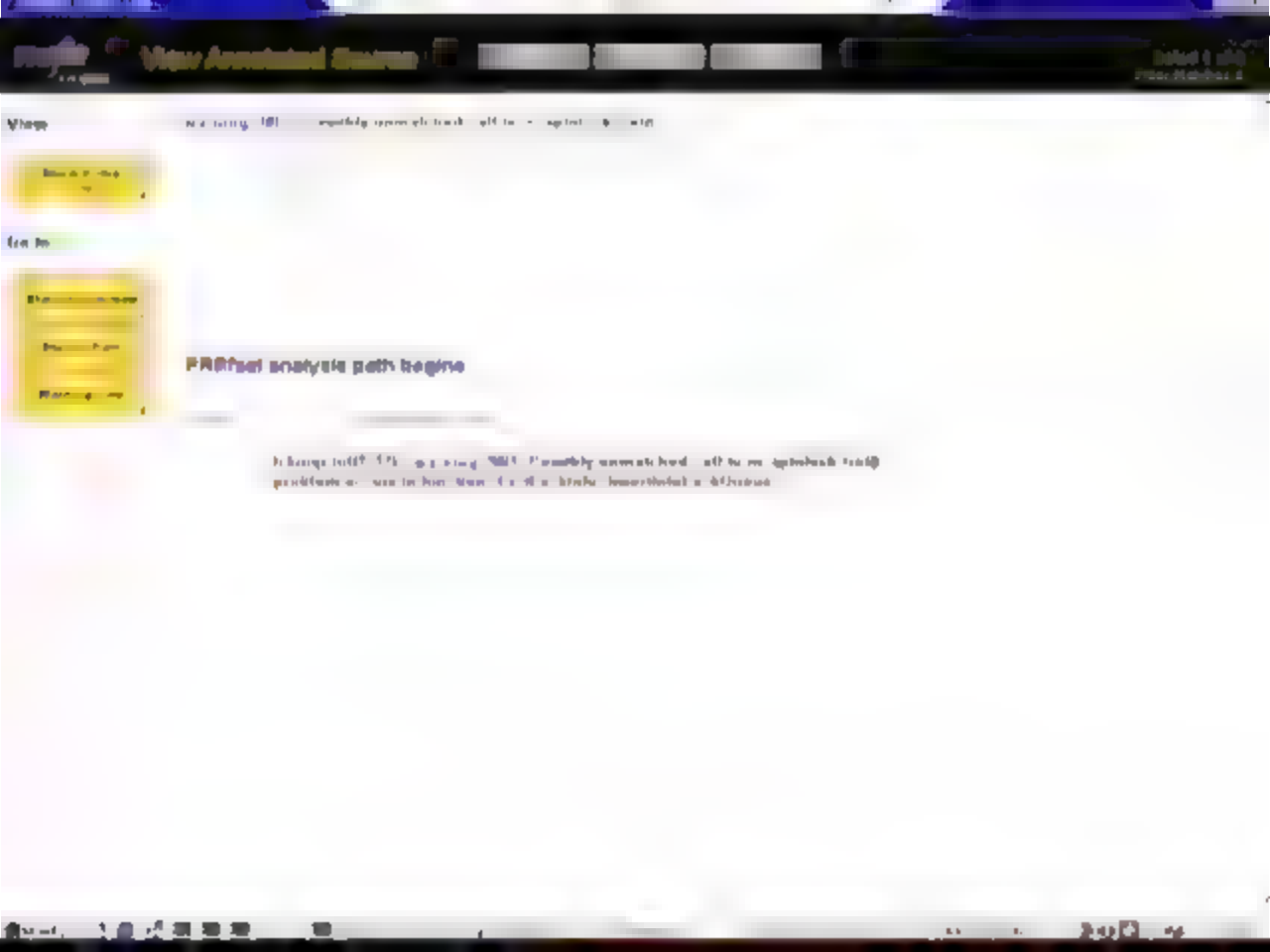


ESP Summary

- Early Implementation for C
 - Verified stdio usage (15 files) in 150K LOC program
- Extend to full C++ and improve speed
- Hypothesis
 - For many (protocols) errors, scalable analysis is sufficiently precise
 - Intricate usage patterns are difficult for programmers, as well as tools







View



Like to



PREVIEW ENGLISH PART 2

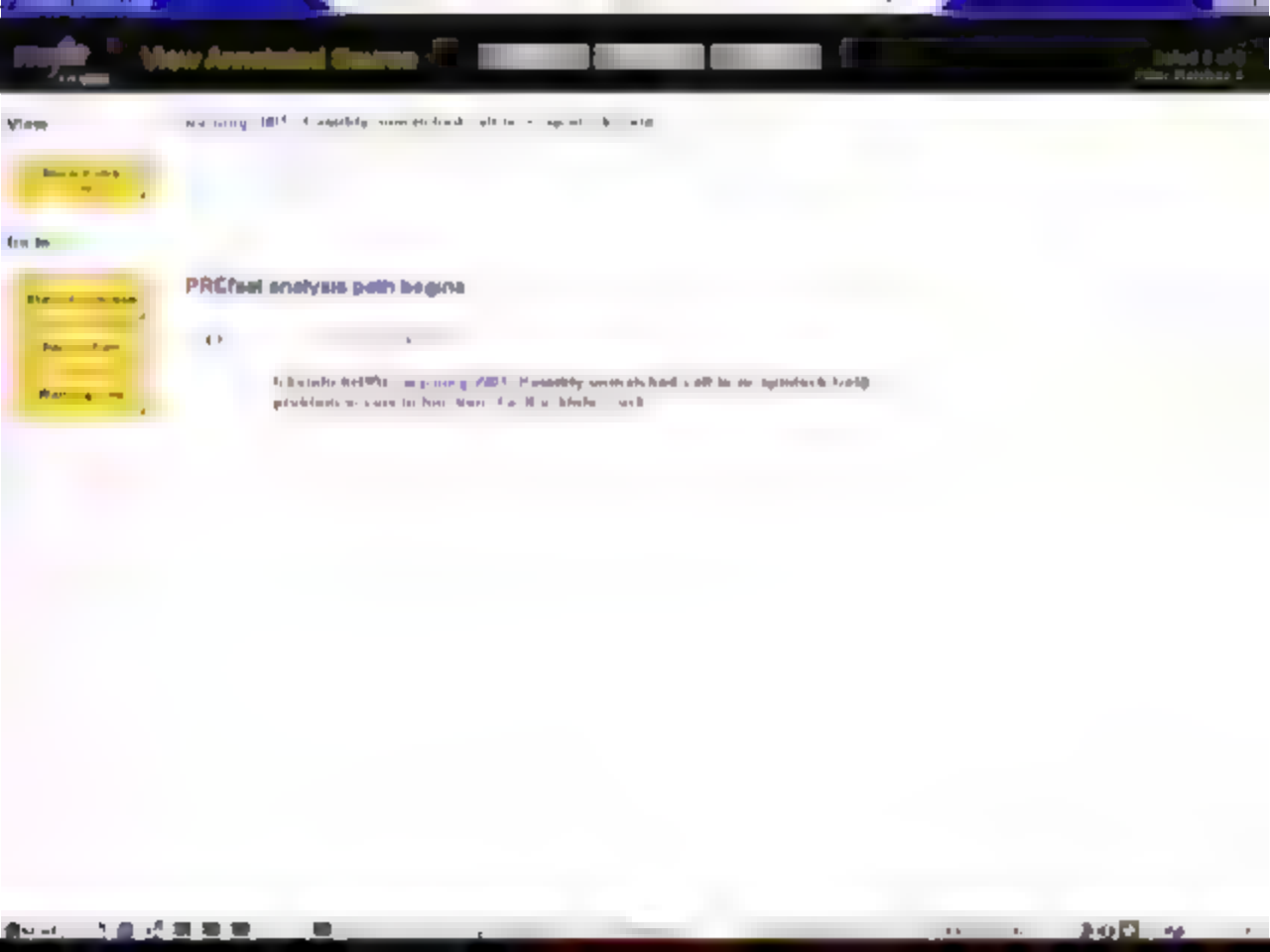
100%

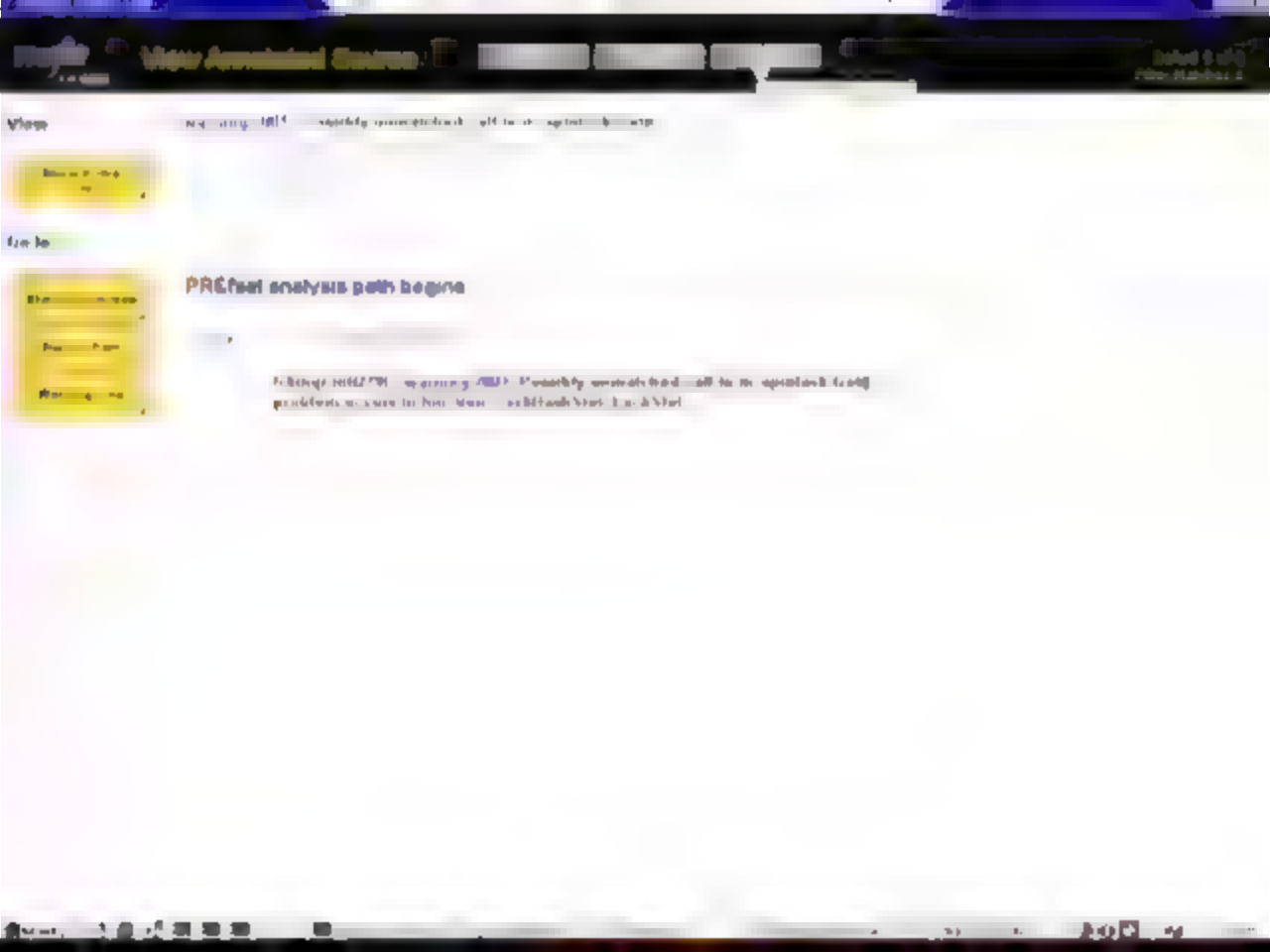
In English (1985.7.12)  I personally cannot find out the English part 2
 part 2 (the part 2 of the English part 2) is a part of the English part 2.



PNEUMONIA: PATHOGENS

Pneumonia is a lung infection that can be caused by a variety of pathogens, including bacteria, viruses, and fungi. The most common cause of pneumonia is bacteria, which can enter the lungs through the mouth or nose. Viruses can also cause pneumonia, and are often the cause of the common cold and flu. Fungi can cause pneumonia in people with weakened immune systems.





View

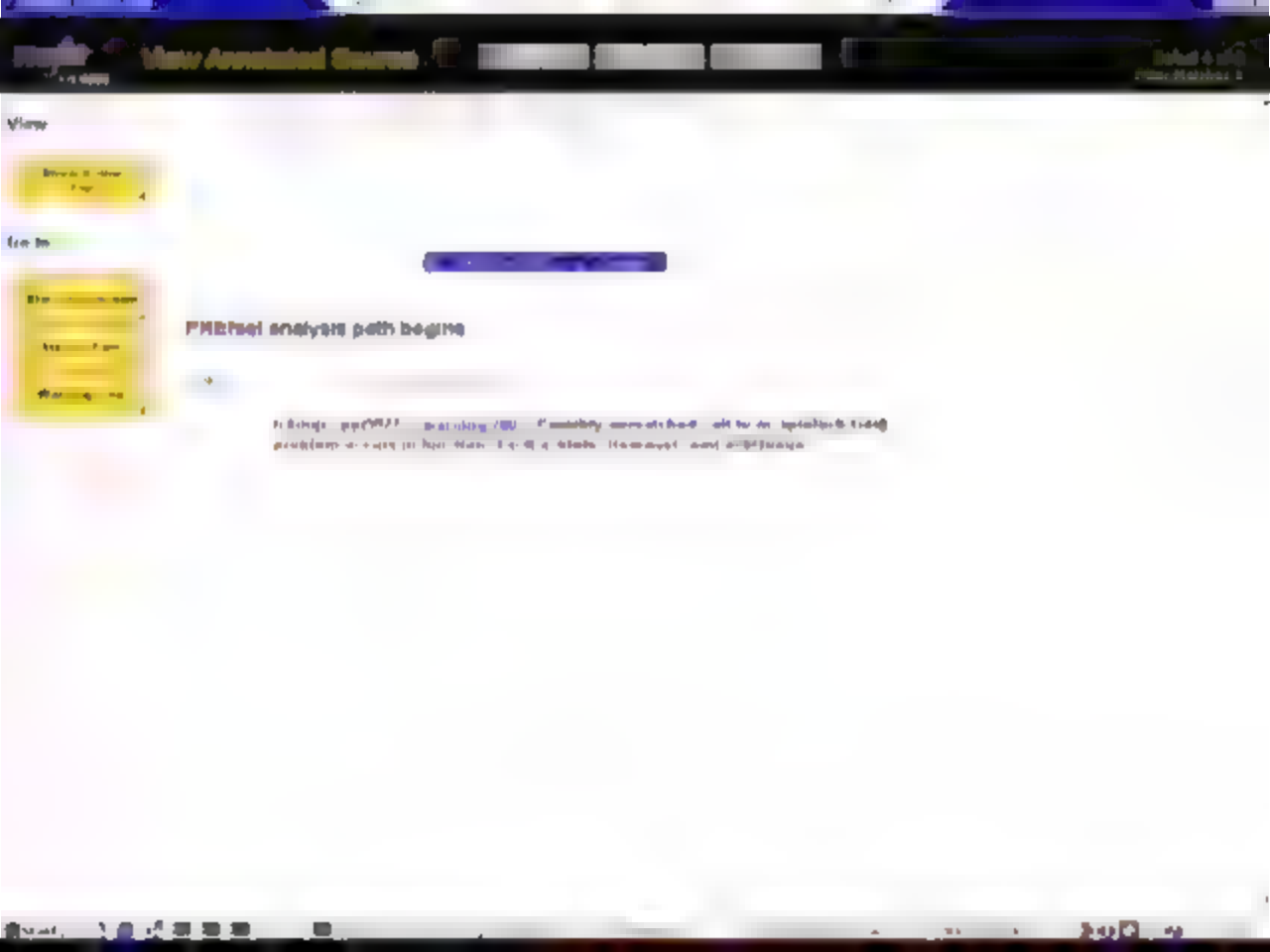
PRCnet analysis path begins

Go to

PRCnet analysis path begins

PRCnet analysis path begins





Slide

Failure Mode

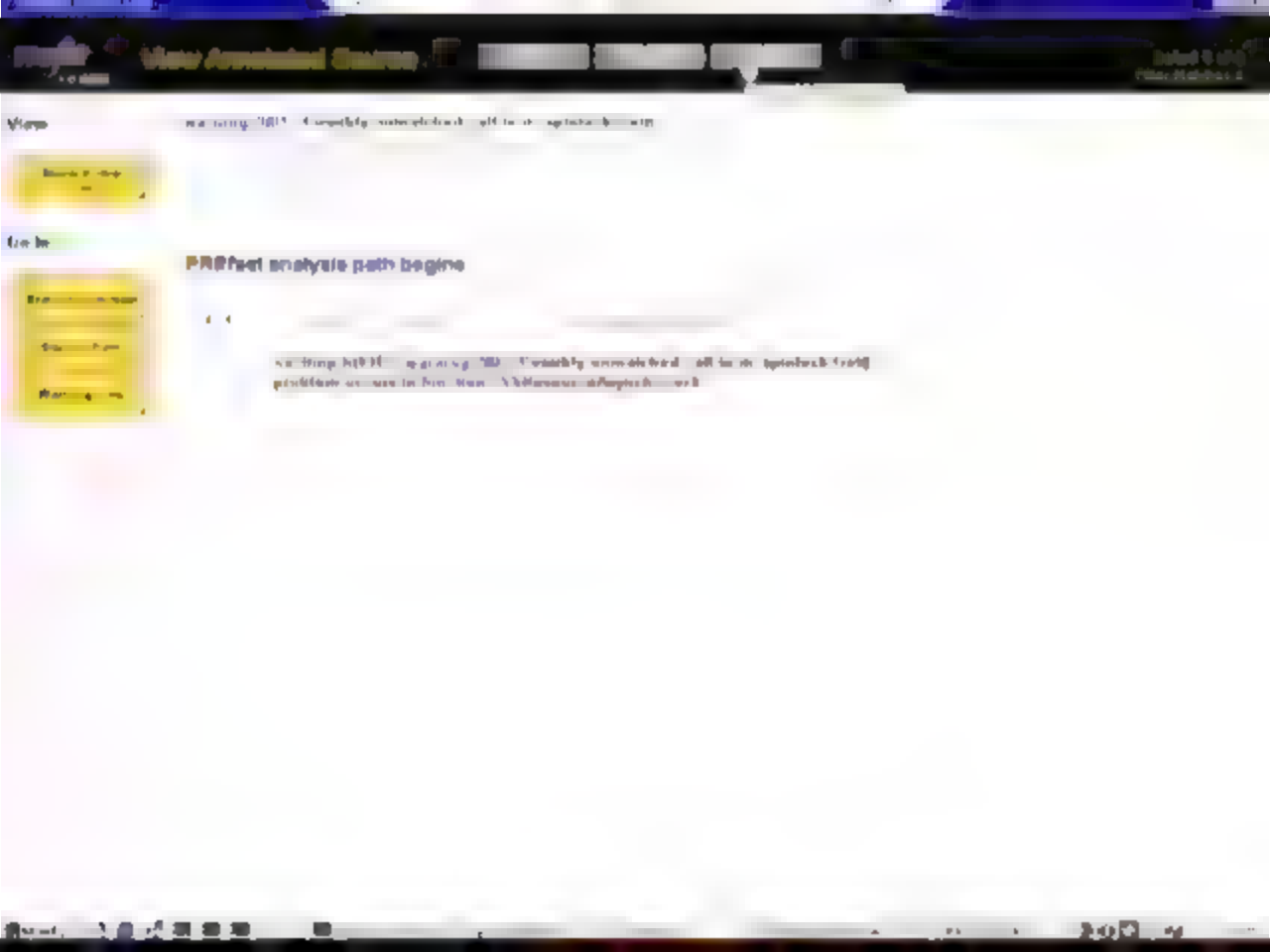
Effect

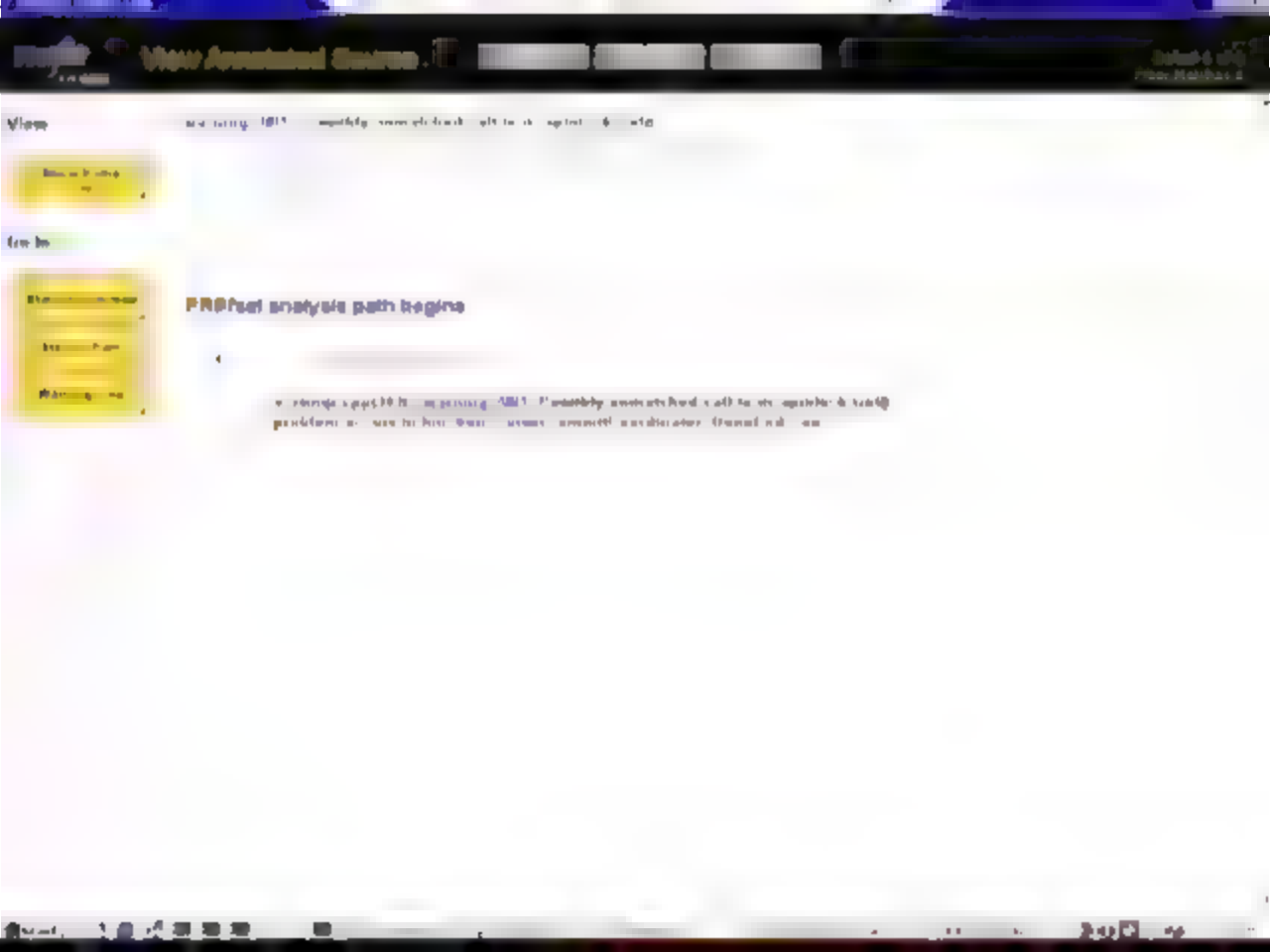
Failure Cause

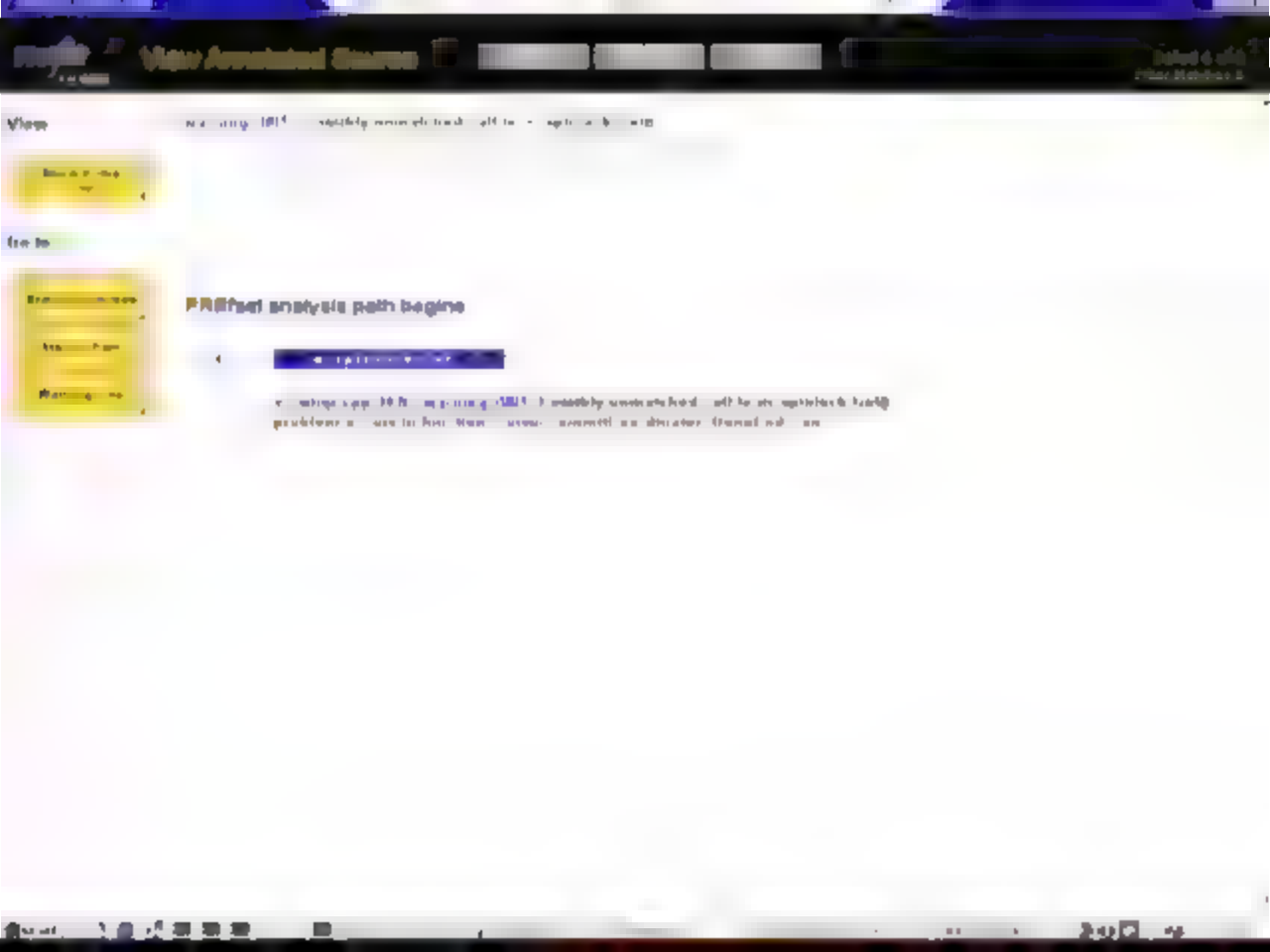
FMECA analysis path begins

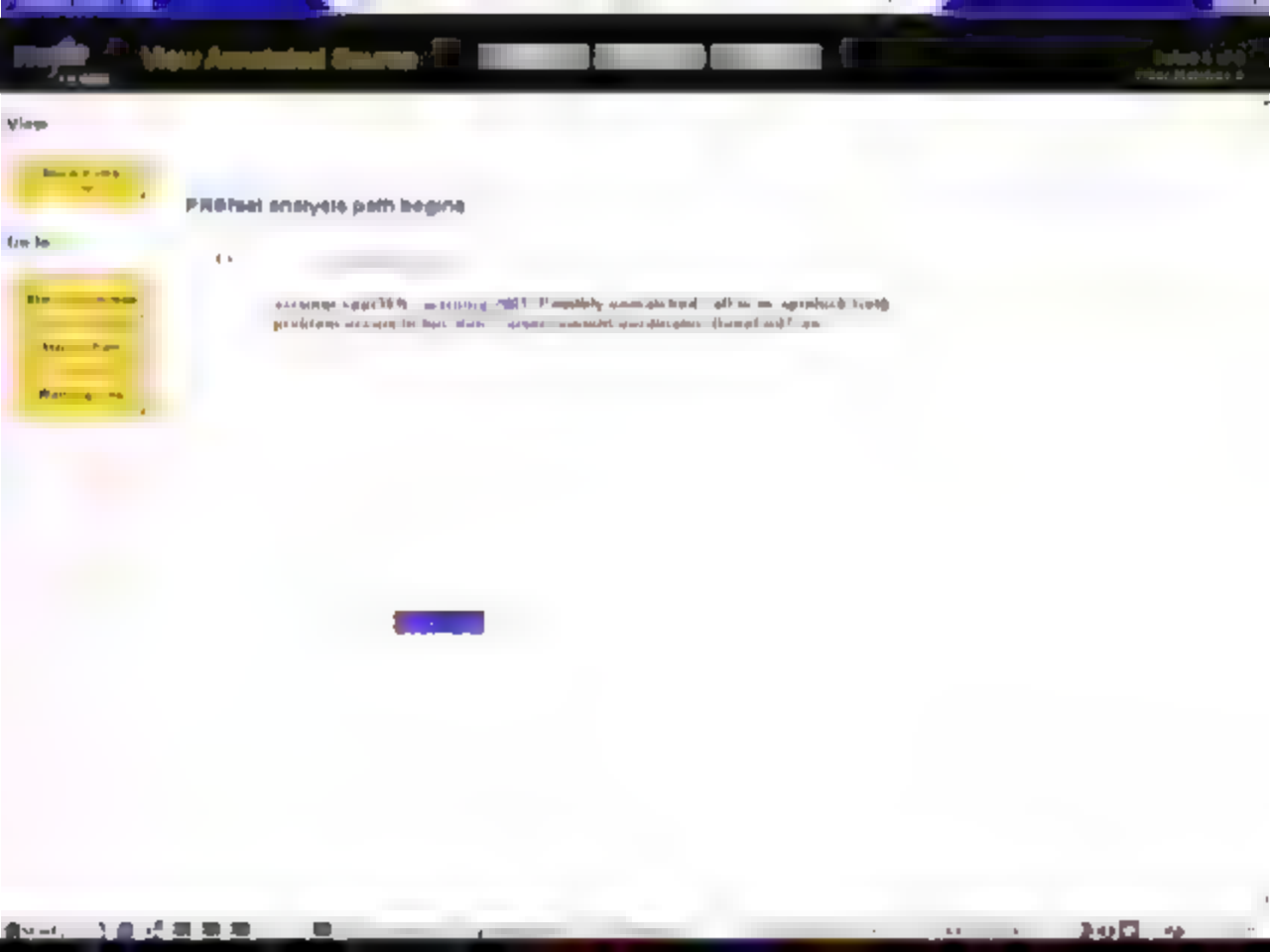
FMECA analysis path begins
Failure Mode
Failure Effect
Failure Cause

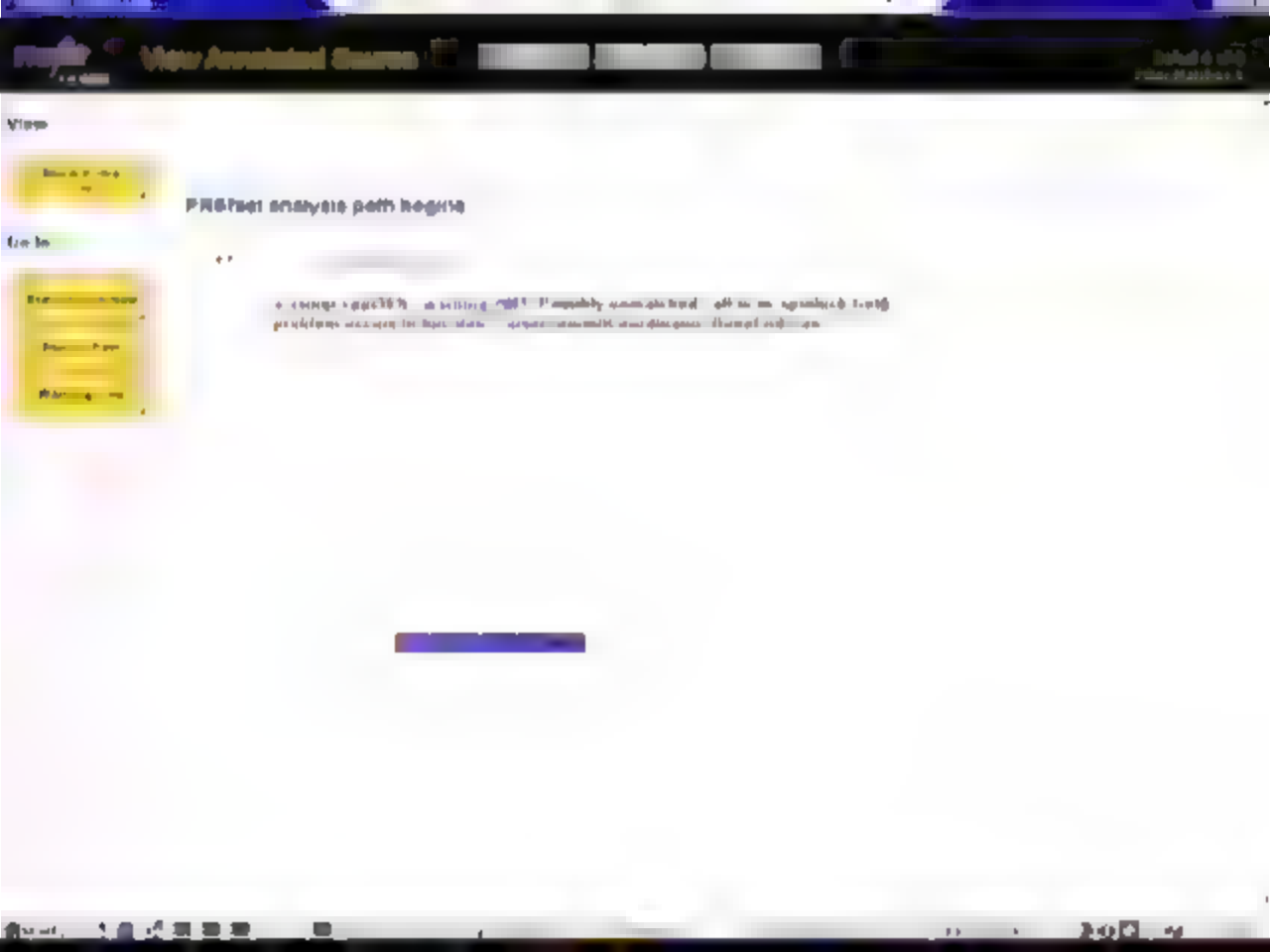


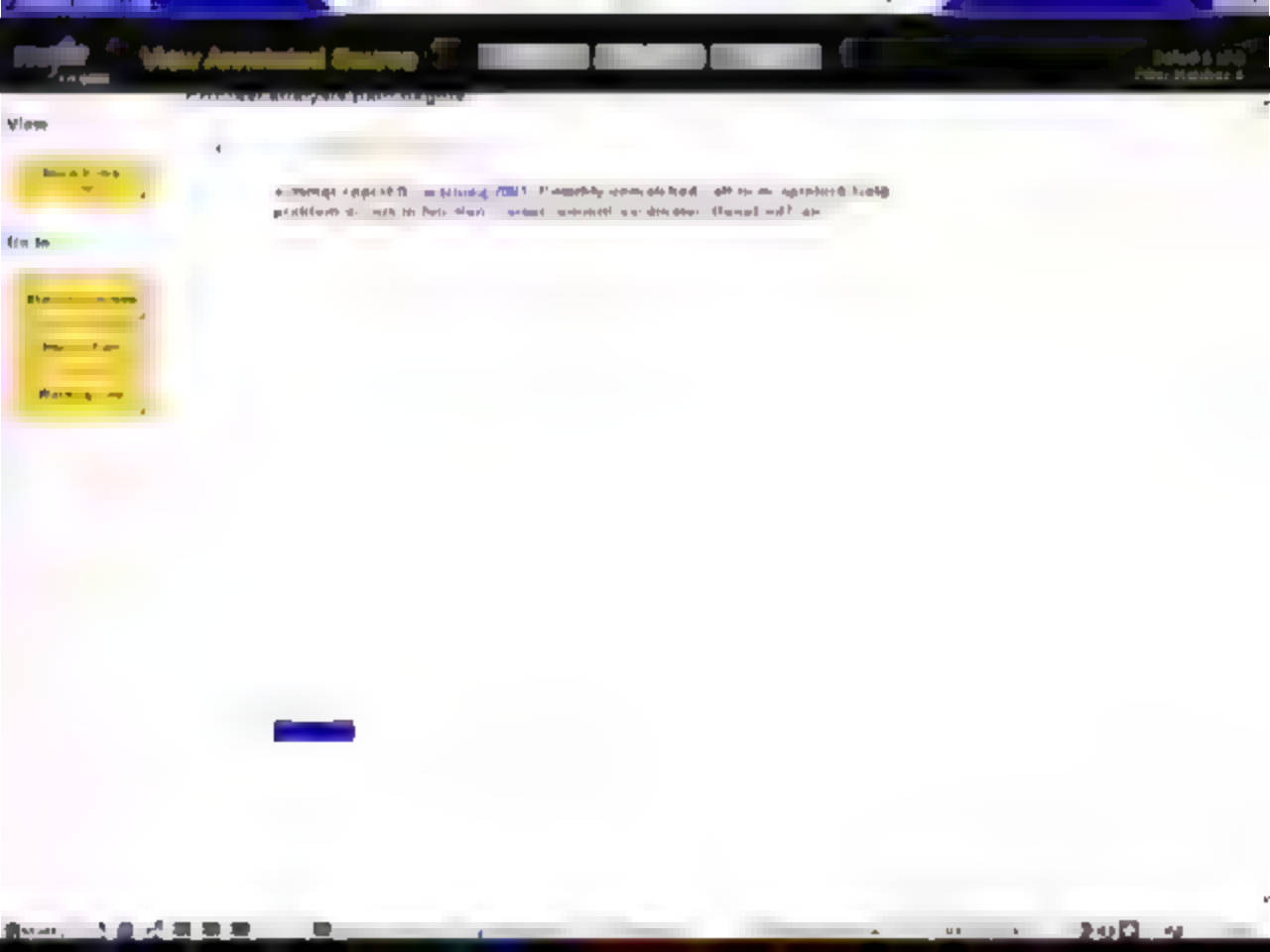












Concurrency

- Parallel and concurrent programs are hard to write, debug, test, modify, ...
 - Everyone is going to be writing many more of them
 - Cheap SMPs and .net
- Asynchronous programming
 - Message-passing (non-RPC) communication
 - Hallelorn, Inigo, ...
 - BizTalk
- Difficult behavioral properties
 - Deadlock-freedom
 - Communication progress
 - Message understood
- Few tools

Sharpie Example

```
// Declaration of client
operation client(In ch: x, Out ch: w) @ x? => w!

// Implementation of server
server (In ch: x, Out ch: y) @ x? => y!
{
  select x?O =>
  {
    async c, d. client(c, d)
    in
    select d?O => y!
  }
}
```

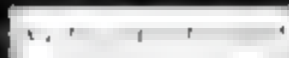
Sharpie Example

```
// Declaration of client  
operation client(In ch: x, Out ch: w) @ x?  $\rightarrow$  w!
```



```
// Implementation of server  
server (In ch: x, Out ch: y) @ x?  $\rightarrow$  y!
```

```
{  
  select x?O  $\rightarrow$   
    async c_d.client(c_d)  
  in  
    select d?O  $\rightarrow$  y!  
}
```



Sharpie Example

```
// declaration of client  
operation client(In ch: z, out ch: w) @ z? -> w!
```

Receive on z
Send on w

```
// implementation of server  
server (In ch: x, out ch: y) @ x? -> y!  
{
```

```
  select x?() ->  
    async c, d, client(c, d),  
  in  
    select d?() -> y!
```

Async call on client

Receive on d

```
}
```

Security Analysis

- Many security flaws exploit information flow

```
UserInput (s1) ;  
...  
CallToKernel (v) ;
```



- Value flow analysis
 - Determines where value comes from or goes to during execution
 - "Can the value of string s1 flow to variable v during execution?"
 - Previously, interprocedural analysis was imprecise or expensive

GOLF (Manuvir Das, Manuel Fahndrich, Jakob Rehof)

- Generalized One Level Flow
 - Whole-program value flow engine
- Scalable, context-sensitive, conservative analysis
 - Handles function calls accurately (context-sensitive)
 - Efficient (MLOC in seconds)
 - Precise (experimental evidence)
- Algorithms incorporated into PREfix
- Basis for family of security tools

SPT Summary

■ SPT approach

- Programmer/tester writes partial specifications
- Tools ensure code follows specs
 - SLAM: extremely precise analysis
 - ESP: trade precision for scalability
 - Vault: integrate approach into language
 - Behavior!; concurrency
 - GOLF: value flow problems

■ Why so many projects?

- Partial verification, not total correctness